INF6032 Big Data Analytics

240175267

Coursework Report

Analysis of Wikipedia and MAGPIE Datasets using Big Data Techniques

Word Count: 3284

Description of Required Setup

All analyses were conducted within a standard Databricks interactive cluster environment utilising PySpark and requiring no non-standard setup procedures. Necessary PySpark libraries and functions were imported at the beginning of the notebook for clarity (Figure 1):

- `pyspark.sql.functions as F` for DataFrame column operations;
- `NGram` and `StopWordsRemover` from `pyspark.ml.feature` for *N*-gram generation and stopword filtering, respectively;
- `Window` from `pyspark.sql.window` for window-based ranking operations.

```
3
from pyspark.sql import functions as F
from pyspark.ml.feature import NGram
from pyspark.ml.feature import StopWordsRemover
from pyspark.sql.window import Window
```

Figure 1.

The primary datasets, `large.csv.gz` and `MAGPIE_unfiltered.jsonl`, were uploaded to `/FileStore/tables/` as per the assignment brief, and then used to load the `large` and `MAGPIE_unfiltered` DataFrames (Figure 2).

```
4
large = spark.read.csv("/FileStore/tables/large.csv.gz", header = True)

MAGPIE_unfiltered = spark.read.json("/FileStore/tables/MAGPIE_unfiltered.jsonl")
```

Figure 2.

Before addressing the questions, the loaded data was inspected using `printSchema()` and `display()` (Figures 3-10).

```
5
large.printSchema()
```

Figure 3.

```
root
 |-- sentence: string (nullable = true)
 |-- source: string (nullable = true)
```

Figure 4.

```
6
display(large.limit(10))
```

Figure 5.

(1) Spark Jobs

Table

	sentence	source
1	"The specific epithet ""seemannii"" refers to someone with the surname 'Seemann	' in many cases it's botanist Berthold Carl Seemann (1825–1871)."
2	Adult and pediatric nurse practitioner programs began in 1971.	pages_articles24
3	"He received the ""Naim Frashëri"" award from the Albanian presidency."	pages_articles24
4	He competed for Germany in the 2018 Winter Olympics.	pages_articles24
5	Despite an increase in the number of votes, Fan failed to win the re-election.	pages_articles24
6	After the match, Cabana challenged Aries to a steel cage match for the title at Third Anniversary Celebration.	pages_articles24
7	Alexander Moissi (; ; 2 April 1879 – 22 March 1935) was an Austrian stage actor (and occasional film actor) of Albanian origin.	pages_articles24
8	His brother is the biomathematician Wolfgang Alt.	pages_articles24
9	› The frontage, refurbished in 2017, displays the carved door crowned with a curved fan light and several cartouches, rosettes...	pages_articles24
10	Tanaka was born in New York City, U.S. on 23 November 1986.	pages_articles24

10 rows | 2.00s runtime

Figure 6.

Figure 7.

```
root
 |-- confidence: double (nullable = true)
 |-- context: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- document_id: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- id: long (nullable = true)
 |-- idiom: string (nullable = true)
 |-- judgment_count: long (nullable = true)
 |-- label: string (nullable = true)
 |-- label_distribution: struct (nullable = true)
 |    |-- ?: double (nullable = true)
 |    |-- f: double (nullable = true)
 |    |-- i: double (nullable = true)
 |    |-- l: double (nullable = true)
 |    |-- o: double (nullable = true)
 |-- non_standard_usage_explanations: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- offsets: array (nullable = true)
 |    |-- element: array (containsNull = true)
 |    |    |-- element: long (containsNull = true)
```

Figure 8.

Figure 9.

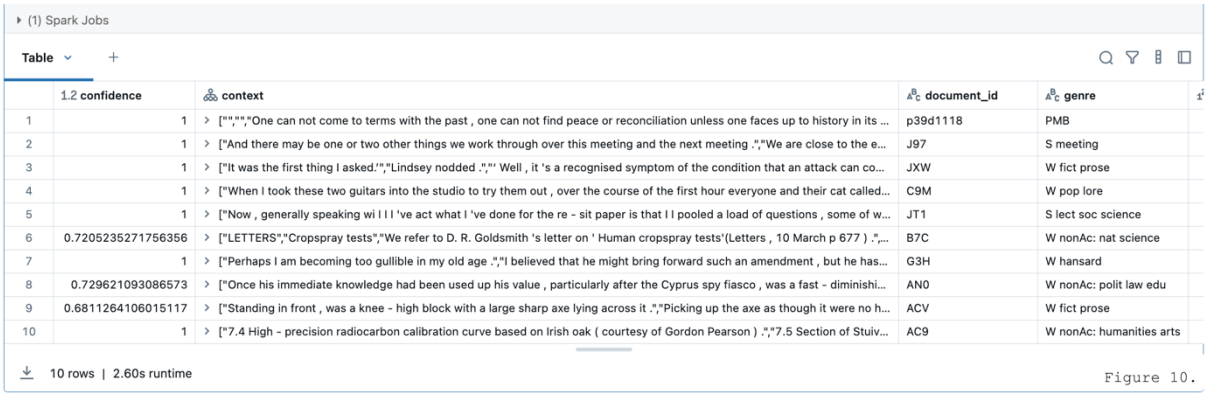


Figure 10.

Data Cleaning and Preparation

The strategy for data cleaning and preparation was guided by the provided test datasets (`small.csv.gz` and `medium.csv.gz`) and their corresponding expected results. Initial implementations of the solutions for each question were developed without extensive global data cleaning steps applied. These implementations were then validated against `small.csv.gz` and `medium.csv.gz`. The outputs precisely matched the 'Past results file', indicating that the baseline level of data cleanliness was sufficient to correctly address the questions.

Question 1.

Assumption:

The `sentence` column in `large.csv.gz` accurately represents individual sentences, where uniqueness is determined by the exact, raw string, including variations in case, punctuation and leading and trailing whitespace. This approach was validated against the test datasets (`small.csv.gz`, `medium.csv.gz`), yielding the expected results.

Implementation:

Distinct sentence count was calculated using a PySpark DataFrame operation. `countDistinct()` was applied to the `sentence` column with a `select()` transformation. The result was then extracted using `.first()[0]` (Figure 11).

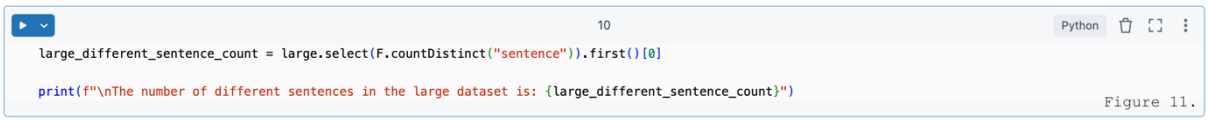


Figure 11.

Result:

The calculated count of distinct sentences was 389,639 (Figure 12).

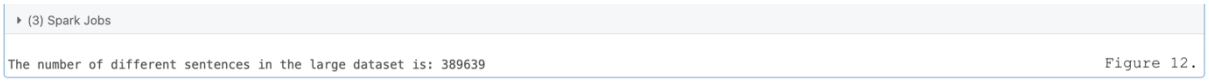


Figure 12.

Discussion, Exploration and Extension:

The baseline count, while useful, is sensitive to text formatting variations such as case and punctuation. To explore this and obtain more semantically meaningful counts, an extension was conducted where text normalisation techniques were applied:

- Conversion of sentences to lowercase;
- Removal of punctuation from sentences;
- Removal of punctuation followed by lowercase conversion.

This exploration involved adding new columns representing the normalised versions of the sentences to the `large` DataFrame and applying the `countDistinct()` aggregation to each of these in a single operation to ensure efficient retrieval of the counts (see Figure 13).

Figure 13.

The exploration (Figure 14) revealed the distinct sentence count to be notably insensitive to the applied normalisation techniques. Compared to the raw count (389,639), lowercasing reduced the count to 389,595 (a decrease of 44; ~0.011%), indicating minimal uniqueness due to case. Punctuation removal had a slightly larger effect, reducing the count to 389,463 (a decrease of 176; ~0.045%). Applying both techniques yielded 389,412 distinct sentences, a total decrease of just 227 (~0.058%).

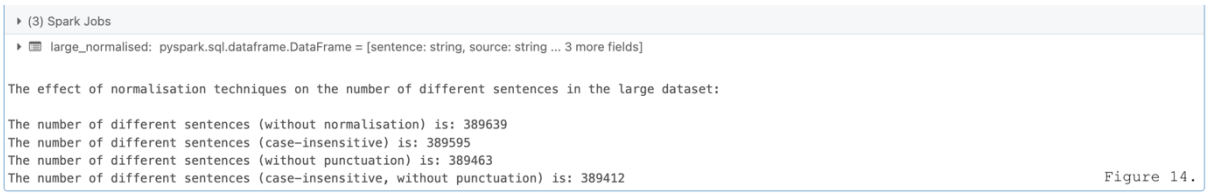


Figure 14.

These findings suggest that sentence uniqueness is primarily driven by word content, rather than formatting variations. Although the impact of normalisation was minimal using this particular dataset, this exploration underscores the critical importance for a client to precisely define ‘different sentences’ based on analytical requirements, particularly for tasks requiring matching, where consistent normalisation is crucial.

Question 2.

Assumptions:

The `sentence` column in the `large` DataFrame represents the textual sentences from the Wikipedia data source.

The specific method using `F.split(F.col("sentence"), "\\s+")` to derive the word count per sentence followed by counting the elements in the resulting array using `F.size()` constitutes the expected basic pre-processing, validated by the results generated matching those provided when tested on `small.csv.gz` and `medium.csv.gz`.

Implementation:

The word counts of the 10 longest sentences were determined by first tokenising them into word arrays using `F.split(F.col("sentence"), "\\s+")`. `word_count` was calculated as the size of these arrays using `F.size(F.col("words"))`. The DataFrame was then sorted by `word_count` descending, the top 10 rows were selected using `.limit(10)` and finally, the `word_count` column was selected and aliased as `Word Count` for display (Figure 15).

Figure 15.

Result:

The analysis to identify the 10 longest sentences revealed lengths at the upper extreme of the distribution. These top 10 sentence lengths range significantly, from 335 words up to an exceptional maximum of 4571 words for the single longest sentence (Figure 16).

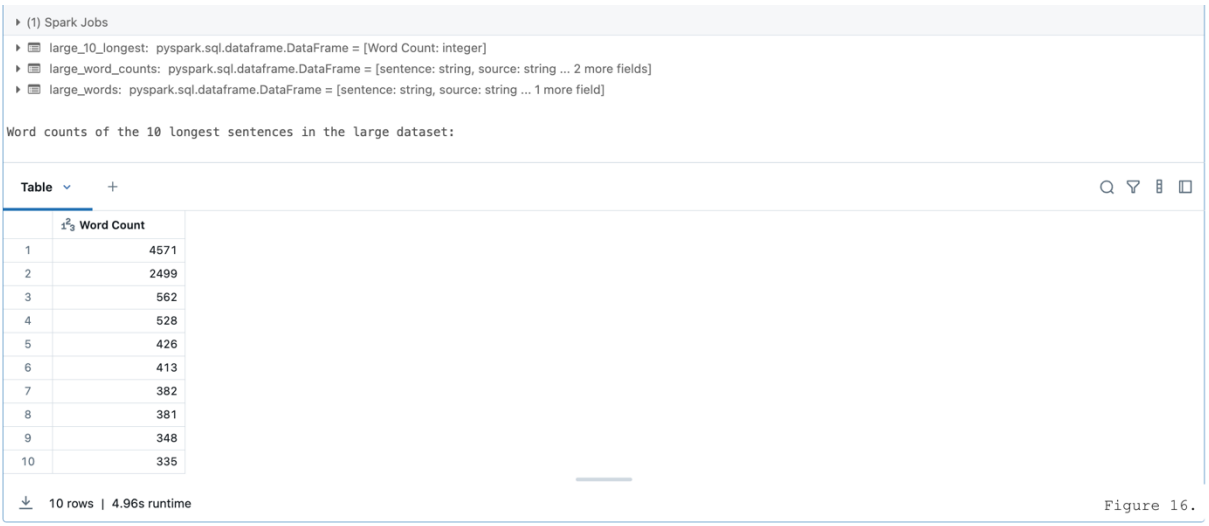


Figure 16.

Discussion, Exploration and Extension:

The word counts of the 10 longest sentences highlight instances of extreme verbosity, suggesting unusually detailed content or a preference for lengthy sentence structures. These extreme values motivate further analysis to contextualise them within the overall sentence length distribution. To this end, descriptive statistics (mean, standard deviation, maximum and minimum) were calculated for both all sentences and the 10 longest sentences, using PySpark's aggregate functions `F.mean()`, `F.stddev()`, `F.max()` and `F.min()`, applied to the respective `word_count` data, with the results extracted using `.first()` (Figure 17).

```
16                                                                    Python

large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_word_counts = large_words.withColumn("word_count", F.size("words")).orderBy(F.desc("word_count"))

large_overall_stats = (large_word_counts.select(
    F.mean("word_count").alias("mean_word_count_overall"),
    F.stddev("word_count").alias("stddev_word_count_overall"),
    F.max("word_count").alias("max_word_count_overall"),
    F.min("word_count").alias("min_word_count_overall")
    ).first()
)

large_10_longest = large_word_counts.select("word_count").limit(10)

large_10_longest_stats = (large_10_longest.select(
    F.mean("word_count").alias("mean_word_count_10_longest"),
    F.stddev("word_count").alias("stddev_word_count_10_longest"),
    F.max("word_count").alias("max_word_count_10_longest"),
    F.min("word_count").alias("min_word_count_10_longest")
    ).first()
)

print("\nOverall sentence statistics for the large dataset:\n")
print(f"Mean word count: {large_overall_stats['mean_word_count_overall']:.2f}")
print(f"Standard deviation of word count: {large_overall_stats['stddev_word_count_overall']:.2f}")
print(f"Maximum word count: {large_overall_stats['max_word_count_overall']}")
print(f"Minimum word count: {large_overall_stats['min_word_count_overall']}")

print("\n\n10 longest sentences statistics for the large dataset:\n")
print(f"Mean word count: {large_10_longest_stats['mean_word_count_10_longest']:.2f}")
print(f"Standard deviation of word count: {large_10_longest_stats['stddev_word_count_10_longest']:.2f}")
print(f"Maximum word count: {large_10_longest_stats['max_word_count_10_longest']}")
print(f"Minimum word count: {large_10_longest_stats['min_word_count_10_longest']}")
```

Figure 17.

Comparing these statistics (Figure 18), provides valuable insights into sentence length distribution within the `large` DataFrame. The overall mean word count (19.04, SD 13.27) indicates generally modest sentence lengths, although the range spans from 1 to an extraordinary 4571 words. In stark contrast to this, the 10 longest sentences exhibit a dramatically different profile, with a mean word count of 1044.50 and SD of 1402.39, indicating immense variability even among these exceptionally long sentences.

```
▸ (5) Spark Jobs
▸ large_10_longest: pyspark.sql.dataframe.DataFrame = [word_count: integer]
▸ large_word_counts: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
▸ large_words: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

Overall sentence statistics for the large dataset:

Mean word count: 19.04
Standard deviation of word count: 13.27
Maximum word count: 4571
Minimum word count: 1


10 longest sentences statistics for the large dataset:

Mean word count: 1044.50
Standard deviation of word count: 1402.39
Maximum word count: 4571
Minimum word count: 335
```

Figure 18.

This statistical context is critical for a client, demonstrating that while typical sentences are relatively short, the dataset contains a significantly long tail with extreme outliers, such as the 4571-word sentence. The pronounced variability among even the top 10 longest sentences further highlights this. An awareness of these characteristics is vital for designing robust text processing pipelines and developing strategies for handling these exceptionally long sentences in subsequent analyses.

Question 3.

Assumptions:

The `sentence` column in the `large` DataFrame provides an accurate representation of individual sentences.

Words are tokenised from these sentences using `F.split(F.col("sentence"), "\\s+")` which splits by whitespace. Punctuation therefore remains attached to the resulting word tokens.

The methodology employed for word tokenisation, bigram generation and calculating their average count is considered the expected basic pre-processing and analytical approach for this task. This is supported by the successful replication of the provided results when testing against `small.csv.gz` and `medium.csv.gz`.

Implementation:

The average number of bigrams per sentence was calculated by first tokenising sentences into word arrays using `F.split(F.col("sentence"), "\\s+")`. The `NGram` transformer (`n=2`, `inputCol="words"`, `outputCol="bigrams"`) from the `pyspark.ml.feature` module then generated arrays of bigram strings. The average number of bigrams per sentence was then computed by applying `F.avg(F.size(F.col("bigrams")))` and extracting the result using the `.first()[0]` action (Figure 19).

```
18                                                                    Python
large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_bigram_transformer = NGram(n = 2, inputCol = "words", outputCol = "bigrams")
large_bigrams = large_bigram_transformer.transform(large_words)

large_average_bigrams = large_bigrams.select(F.avg(F.size(F.col("bigrams")))).first()[0]

print(f"\nThe average number of bigrams per sentence in the large dataset is: {large_average_bigrams}")
```

Figure 19.

Result:

The PySpark execution to calculate the average number of bigrams per sentence in the `large` DataFrame produced an output of 18.04 bigrams (Figure 20).

```
▸ (2) Spark Jobs
▸ large_bigrams: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
▸ large_words: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

The average number of bigrams per sentence in the large dataset is: 18.0365125
```

Figure 20.

Discussion, Exploration and Extension:

The result indicating an average of 18.04 bigrams per sentence in the `large` DataFrame provides a useful metric. The average is comparable to the overall average word count per sentence calculated in the Question 2 extension (19.04), **minus one**, which is expected, as sentences with *N* words can form a maximum of *N*-1 bigrams. The average bigram count suggests that sentences in the dataset are generally long enough to contain a substantial number of sequential two-word pairings, making them a potentially rich source for analyses based on these features. A logical further exploration is to investigate the prevalence of slightly longer, three-word sequences (trigrams). This helps to gauge how quickly the density of common *N*-gram phrase structures decreases as *N* increases, offering further insight into sentence complexity and common phrase lengths. The methodology for calculating the average number of trigrams per sentence mirrored that for bigrams with the `NGram` transformer from `pyspark.ml.feature` configured with `n=3` (to specify trigrams) (Figure 21).

Figure 21.

The analysis established that the average number of trigrams per sentence was 17.04. This average (Figure 22) is, as expected, lower than the average of 18.04 bigrams per sentence and represents approximately one fewer three-word sequence per sentence on average compared to two-word sequences (a 5.5% decrease). This illustrates that, consistent with the general nature of language, longer specific *N*-grams are generally less frequent as a feature within sentences than shorter *N*-grams due to the increasing specificity of longer sequences.

Figure 22.

For a client, understanding the fall-off rate in *N*-gram prevalence (from an average of 18.04 for bigrams to 17.04 for trigrams and likely further still for longer *N*-grams) is important as it can inform tasks such as feature engineering for machine learning models, where trigrams might offer more specific contextual information than bigrams, and in the development of pattern-matching systems where the expected density of different *N*-gram lengths impacts efficiency and effectiveness.

Question 4.

Assumptions:

Words are tokenised from sentences using `F.split(F.col("sentence"), "\\s+")` (preserving original case and punctuation) and bigrams are subsequently generated as consecutive pairs of these words using the `NGram(n=2)` transformer.

The frequency of each bigram is determined by the total count of its exact string occurrences as derived from the above generation process.

This combined approach to word tokenisation, bigram formation and frequency counting is considered the basic pre-processing requirement for this analysis. This is validated by the successful replication of the provided results for Question 4 when testing against the test datasets.

Implementation:

To identify the 10 most frequent bigrams, the methodology detailed in Question 3 was followed for initial sentence tokenisation and per-sentence bigram array generation. These bigram arrays were then processed using `F.explode(F.col("bigrams"))` (aliased as `Bigram`), followed by `groupBy("Bigram").count()` to calculate frequencies. The top 10 were determined by ordering by count descending (`orderBy(F.col("count").desc())`) and applying `.limit(10)`. A final `select()` operation formatted the output as `Bigram` and `Count` columns (Figure 23).

22 Python

```
large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_bigram_transformer = NGram(n = 2, inputCol = "words", outputCol = "bigrams")
large_bigrams = large_bigram_transformer.transform(large_words)

large_exploded_bigrams = large_bigrams.select(F.explode(F.col("bigrams")).alias("Bigram"))

large_bigram_counts = large_exploded_bigrams.groupBy("Bigram").count()

large_most_freq_bigrams = (large_bigram_counts.orderBy(F.col("count").desc()).limit(10).select(
                                                                                    F.col("Bigram"),
                                                                                    F.col("count").alias("Count")
                                                                                    )
)

print("\nThe 10 most frequent bigrams in the large dataset:\n")
large_most_freq_bigrams.display()
```

Figure 23.

In addition to the DataFrame method, an alternative implementation using Spark's RDD API was also developed to calculate the top 10 most frequent bigrams. This approach involved manual tokenisation, bigram pairing and distributed counting, producing identical results (Figure 24). The DataFrame API with the `NGram` transformer was chosen as the primary technique in this report due to its conciseness.

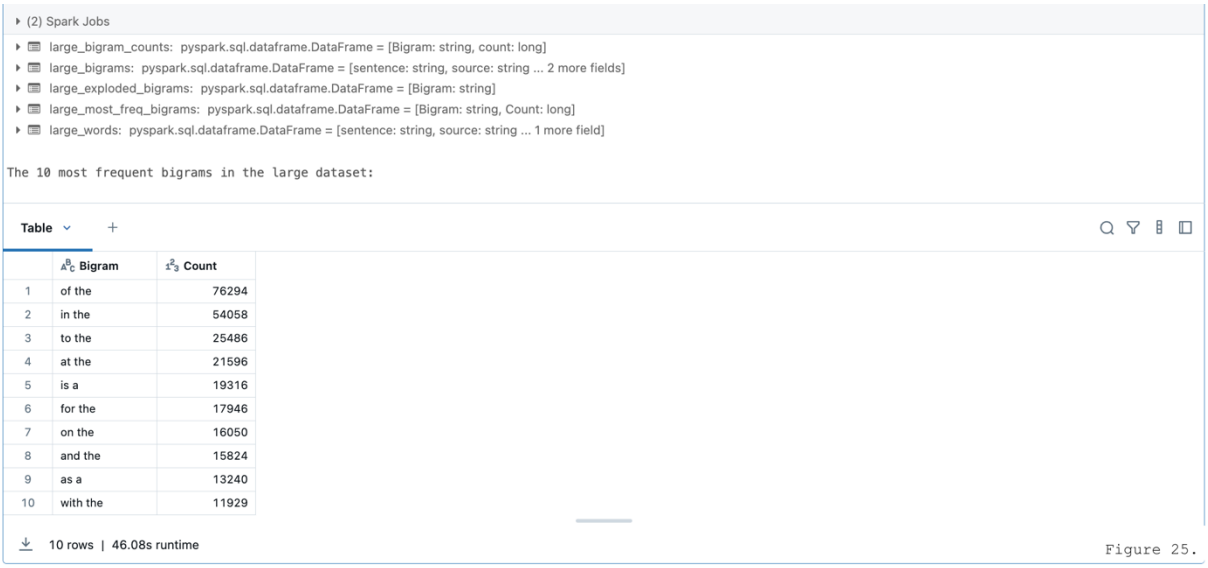
24 Python

```
large_sentence_rdd = large.select("sentence").rdd.map(lambda row: row.sentence)

large_bigram_rdd = large_sentence_rdd.flatMap(
    lambda sentence:
        (lambda words: [words[i] + " " + words[i+1] for i in range(len(words) - 1)] if len(words) >= 2 else [])
        (sentence.split())
)

large_bigram_pairs_rdd = large_bigram_rdd.map(lambda bigram: (bigram, 1))

large_bigram_counts_rdd = large_bigram_pairs_rdd.reduceByKey(lambda x, y: x + y)

large_sorted_bigram_counts_rdd = large_bigram_counts_rdd.sortBy(lambda item: item[1], ascending = False)

large_10_most_freq_bigrams = large_sorted_bigram_counts_rdd.take(10)

print("\nThe 10 most frequent bigrams in the large dataset\n(Alternative implementation using RDD):\n")
spark.createDataFrame(large_10_most_freq_bigrams, ["Bigram", "Count"]).display()
```

Figure 24.

Result:

PySpark DataFrame API execution identified the top 10 most frequent bigrams and their counts, detailing common grammatical pairings such as ‘of the’ (76,294 occurrences) and ‘in the’ (54,058 occurrences) (Figure 25).



(2) Spark Jobs

large_bigram_counts: pyspark.sql.dataframe.DataFrame = [Bigram: string, count: long]
large_bigrams: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
large_exploded_bigrams: pyspark.sql.dataframe.DataFrame = [Bigram: string]
large_most_freq_bigrams: pyspark.sql.dataframe.DataFrame = [Bigram: string, Count: long]
large_words: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

The 10 most frequent bigrams in the large dataset:

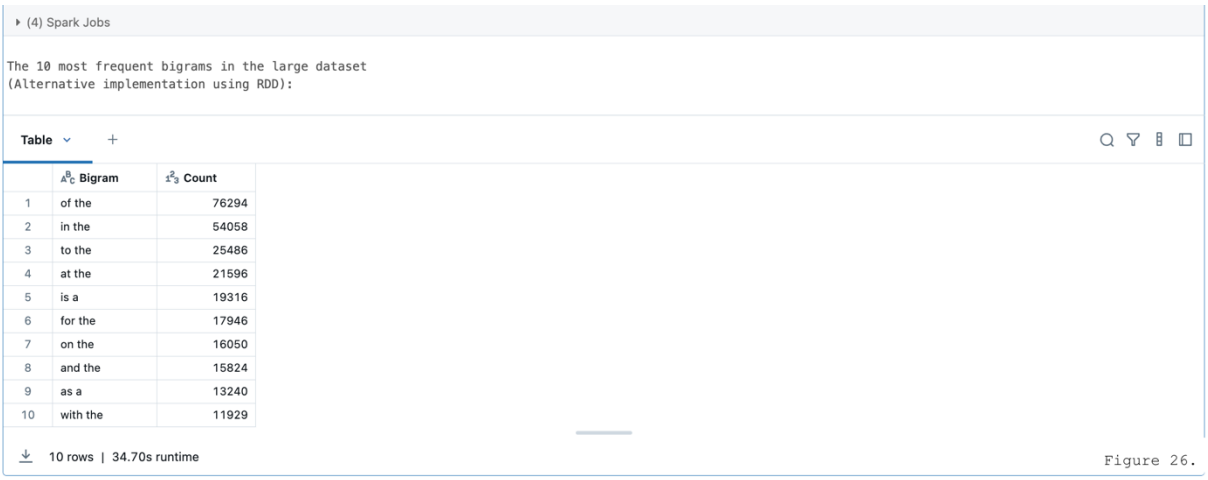
Table +

	Bigram	Count
1	of the	76294
2	in the	54058
3	to the	25486
4	at the	21596
5	is a	19316
6	for the	17946
7	on the	16050
8	and the	15824
9	as a	13240
10	with the	11929

10 rows | 46.08s runtime

Figure 25.

Additionally, the identical results produced using Spark’s RDD API are shown (Figure 26).



(4) Spark Jobs

The 10 most frequent bigrams in the large dataset
(Alternative implementation using RDD):

Table +

	Bigram	Count
1	of the	76294
2	in the	54058
3	to the	25486
4	at the	21596
5	is a	19316
6	for the	17946
7	on the	16050
8	and the	15824
9	as a	13240
10	with the	11929

10 rows | 34.70s runtime

Figure 26.

Discussion, Exploration and Extension:

The 10 most frequent bigrams were dominated by pairs such as ‘of the’ (76,294) and ‘in the’ (54,058). While these high frequencies indicate the structural importance of these bigrams in sentence formation, they offer limited direct insight into particular topics or semantic content. For a client, this list highlights common linguistic patterns but also demonstrates that, without further pre-processing, high-frequency *N*-grams may not be the most informative for thematic analysis.

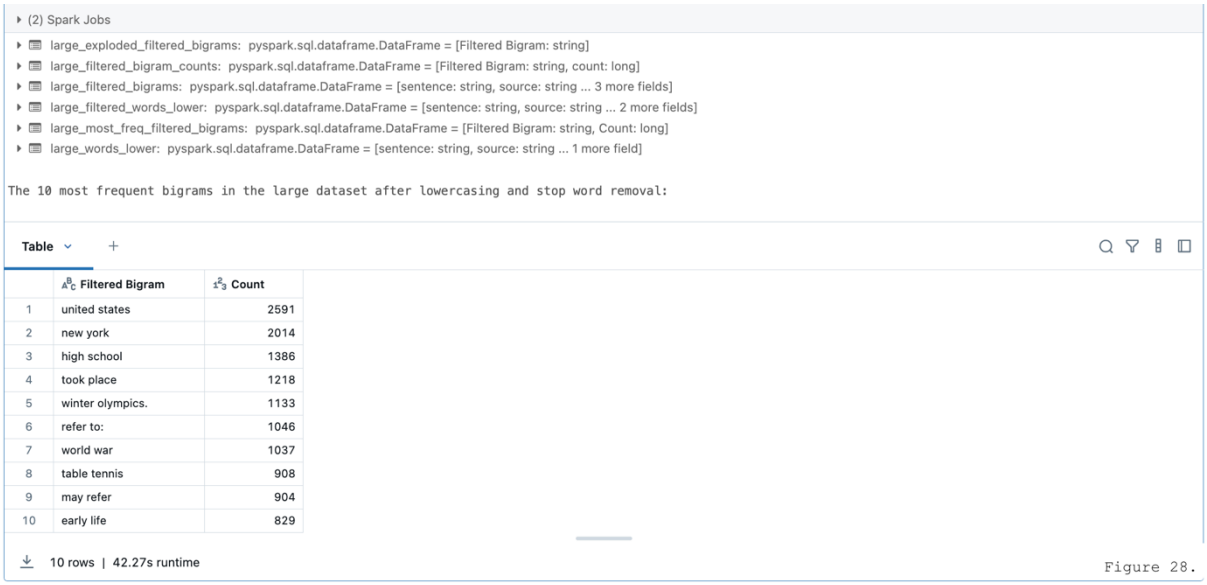
To gain deeper insights beyond analysis of raw bigram frequencies and uncover more content-focussed phrases, several further explorations were conducted. These included analysing the impact of pre-processing (through lowercasing and stopword removal) on frequent bigrams, before extending the *N*-gram analysis to filtered trigrams. The initial exploration focussed on identifying frequent bigrams after applying text normalisation to mitigate the dominance of common grammatical structures. This involved converting all sentences to lowercase and then removing standard English stopwords.

To identify more semantically meaningful frequent two-word phrases, the text was first normalised by converting all sentences to lowercase, followed by removing common stopwords using the `StopWordsRemover` transformer. Bigrams were then generated from these filtered word lists using the `NGram(n=2)` transformer. The subsequent steps of exploding these bigrams, grouping by the bigram string, counting frequencies, ordering by count (descending) and limiting to the top 10 mirrored the approach used for raw bigrams (Figure 27).

```
26                                                                                                    Python

large_words_lower = large.withColumn("words", F.split(F.lower(F.col("sentence")), "\\s+"))

stop_word_remover = StopWordsRemover(inputCol = "words", outputCol = "filtered_words")

large_filtered_words_lower = stop_word_remover.transform(large_words_lower)

large_bigram_transformer = NGram(n = 2, inputCol = "filtered_words", outputCol = "filtered_bigrams")
large_filtered_bigrams = large_bigram_transformer.transform(large_filtered_words_lower)

large_exploded_filtered_bigrams = large_filtered_bigrams.select(F.explode(F.col("filtered_bigrams")).alias("Filtered Bigram"))

large_filtered_bigram_counts = large_exploded_filtered_bigrams.groupBy("Filtered Bigram").count()

large_most_freq_filtered_bigrams = (large_filtered_bigram_counts.orderBy(F.col("count").desc()).limit(10).select(
                                                                    F.col("Filtered Bigram"),
                                                                    F.col("count").alias("Count")
                                                                    )
)

print("\nThe 10 most frequent bigrams in the large dataset after lowercasing and stop word removal:\n")
large_most_freq_filtered_bigrams.display()
```

Figure 27.

Applying these pre-processing steps significantly changed the composition of the most frequent bigrams. The list now included indicative topics such as ‘united states’ (2,591 occurrences) and ‘new york’ (2,014 occurrences), with the full top 10 detailed in Figure 28.



▸ (2) Spark Jobs

▸ large_exploded_filtered_bigrams: pyspark.sql.dataframe.DataFrame = [Filtered Bigram: string]
▸ large_filtered_bigram_counts: pyspark.sql.dataframe.DataFrame = [Filtered Bigram: string, count: long]
▸ large_filtered_bigrams: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 3 more fields]
▸ large_filtered_words_lower: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
▸ large_most_freq_filtered_bigrams: pyspark.sql.dataframe.DataFrame = [Filtered Bigram: string, Count: long]
▸ large_words_lower: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

The 10 most frequent bigrams in the large dataset after lowercasing and stop word removal:

Table +

	Filtered Bigram	Count
1	united states	2591
2	new york	2014
3	high school	1386
4	took place	1218
5	winter olympics.	1133
6	refer to:	1046
7	world war	1037
8	table tennis	908
9	may refer	904
10	early life	829

10 rows | 42.27s runtime

Figure 28.

The top 10 filtered bigrams contrast sharply with the raw bigram analysis, as the additional pre-processing surfaces topical and named entity bigrams such as ‘united states’, ‘high school’ and ‘world war’. Their frequencies (829-2,591 occurrences) are substantially lower than the tens of thousands observed for the top raw bigrams, which is an expected outcome of stopword removal and case normalisation. For a client, this filtered list is likely to be significantly more valuable for tasks such as understanding prominent topics within the Wikipedia dataset, keyword extraction, or as features for text classification, as it prioritises content over grammatical structure. It clearly demonstrates the utility of basic pre-processing in refining *N*-gram analysis for semantic insight.

Further extending the *N*-gram analysis, both raw and filtered trigrams (three-word sequences) were also generated and examined to assess the characteristics of longer phrases. This investigation (Figures 29-32), confirmed that while trigrams can capture more specific concepts, they occur with lower overall frequency. Pre-processing (lowercasing and stopword removal) was similarly effective in highlighting more content-bearing trigrams compared to raw trigram sequences, which were also dominated by common structural words. This comparative *N*-gram analysis - from raw bigrams to filtered bigrams and filtered trigrams - illustrates how choices in pre-processing and *N*-gram length significantly influence the types of linguistic patterns identified as 'most frequent'. While raw *N*-grams reveal common structural elements, filtered *N*-grams are more effective for uncovering topical content. Increasing the *N*-gram length after filtering provides more specific phrases, though typically with lower frequencies. The optimal approach would depend on the client's specific analytical goals.

```
28                                                                                    Python
large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_trigram_transformer = NGram(n = 3, inputCol = "words", outputCol = "trigrams")
large_trigrams = large_trigram_transformer.transform(large_words)

large_exploded_trigrams = large_trigrams.select(F.explode(F.col("trigrams")).alias("Trigram"))

large_trigram_counts = large_exploded_trigrams.groupBy("Trigram").count()

large_most_freq_trigrams = (large_trigram_counts.orderBy(F.col("count").desc()).limit(10).select(
                                                                                    F.col("Trigram"),
                                                                                    F.col("count").alias("Count")
                                                                                    )
)

print("\nThe 10 most frequent trigrams in the large dataset:\n")
large_most_freq_trigrams.display()
```

Figure 29.

```
▸ (2) Spark Jobs
▸ large_exploded_trigrams:  pyspark.sql.dataframe.DataFrame = [Trigram: string]
▸ large_most_freq_trigrams:  pyspark.sql.dataframe.DataFrame = [Trigram: string, Count: long]
▸ large_trigram_counts:  pyspark.sql.dataframe.DataFrame = [Trigram: string, count: long]
▸ large_trigrams:  pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
▸ large_words:  pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

The 10 most frequent trigrams in the large dataset:
```

Table +

	Trigram	Count
1	one of the	3307
2	the University of	3096
3	as well as	2991
4	member of the	2768
5	part of the	2668
6	a member of	2393
7	was born in	2157
8	the end of	1933
9	the United States	1875
10	at the University	1494

10 rows | 44.34s runtime

Figure 30.

```
30                                                                                    Python
large_words_lower = large.withColumn("words", F.split(F.lower(F.col("sentence")), "\\s+"))

stop_word_remover = StopWordsRemover(inputCol = "words", outputCol = "filtered_words")

large_filtered_words_lower = stop_word_remover.transform(large_words_lower)

large_trigram_transformer = NGram(n = 3, inputCol = "filtered_words", outputCol = "filtered_trigrams")
large_filtered_trigrams = large_trigram_transformer.transform(large_filtered_words_lower)

large_exploded_filtered_trigrams = large_filtered_trigrams.select(F.explode(F.col("filtered_trigrams")).alias("Filtered Trigram"))

large_filtered_trigram_counts = large_exploded_filtered_trigrams.groupBy("Filtered Trigram").count()

large_most_freq_filtered_trigrams = (large_filtered_trigram_counts.orderBy(F.col("count").desc()).limit(10).select(
                                                                                    F.col("Filtered Trigram"),
                                                                                    F.col("count").alias("Count")
                                                                                    )
)

print("\nThe 10 most frequent trigrams in the large dataset after lowercasing and stop word removal:\n")
large_most_freq_filtered_trigrams.display()
```

Figure 31.

(2) Spark Jobs

- large_exploded_filtered_trigrams: pyspark.sql.dataframe.DataFrame = [Filtered Trigram: string]
- large_filtered_trigram_counts: pyspark.sql.dataframe.DataFrame = [Filtered Trigram: string, count: long]
- large_filtered_trigrams: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 3 more fields]
- large_filtered_words_lower: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
- large_most_freq_filtered_trigrams: pyspark.sql.dataframe.DataFrame = [Filtered Trigram: string, Count: long]
- large_words_lower: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]

```
The 10 most frequent trigrams in the large dataset after lowercasing and stop word removal:
```

Table +

	Filtered Trigram	Count
1	may refer to:	824
2	early life education.	604
3	world table tennis	582
4	table tennis championships	533
5	national register historic	482
6	register historic places	451
7	professional footballer plays	439
8	listed national register	391
9	new york city	340
10	people surname include:	326

10 rows | 41.32s runtime

Figure 32.

Question 5.

Assumptions:

The `sentence` column in the `large` DataFrame accurately represents the textual sentences from Wikipedia and the `idiom` column in the `MAGPIE_unfiltered` DataFrame contains the string representations of idioms from the `MAGPIE` subset.

A Wikipedia bigram is considered to 'appear in the list of idioms' if its string representation finds a match with an idiom string from the 'prepared `MAGPIE` list'.

The matching criterion between Wikipedia bigrams and `MAGPIE` idioms is an exact, case-sensitive, punctuation-inclusive comparison. This specific comparison method, as part of the overall methodology is considered the expected basic pre-processing and analytical approach. This is validated by the successful matching of the numerical count for Question 5 when testing against `small.csv.gz` and `medium.csv.gz`.

Implementation:

To count the unique Wikipedia bigrams also present in the `MAGPIE` idiom list, per-sentence bigram arrays were first generated from the `large` DataFrame following the methodology detailed in Question 4 (involving `F.split` and `NGram`). A unique list of individual Wikipedia bigrams (aliased as `bigram`) was then obtained using `F.explode(F.col("bigrams"))` followed by `.distinct()`. Separately, distinct idiom strings were extracted from the `MAGPIE_unfiltered` DataFrame's `idiom` column using `.distinct()`. These two distinct sets were then joined using an `inner join` on `F.col("bigram")==F.col("idiom")`. The final count of matching unique bigrams was derived using `.count()` on the joined DataFrame (Figure 33).

```
32                                                                                                                    Python
large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_bigram_transformer = NGram(n = 2, inputCol = "words", outputCol = "bigrams")
large_bigrams = large_bigram_transformer.transform(large_words)

large_exploded_bigrams = (large_bigrams.select(F.explode(F.col("bigrams")).alias("bigram")).distinct())

MAGPIE_unfiltered_idioms = MAGPIE_unfiltered.select("idiom").distinct()

large_matching = large_exploded_bigrams.join(MAGPIE_unfiltered_idioms, F.col("bigram") == F.col("idiom"), "inner")

count = large_matching.count()

print(f"\nNumber of bigrams in the large dataset matching MAGPIE_unfiltered idioms: {count}")
```

Figure 33.

An alternative implementation using PySpark SQL was also created to address Question 5 and confirmed to output an identical result. This SQL approach can be reviewed in Figure 34. The DataFrame API was selected for detailed presentation due to its consistency with the other solutions.

Figure 34.

Result:

The PySpark execution to determine how many unique Wikipedia bigrams also appear in the `MAGPIE` idiom list yielded a count of 67 (Figure 35).

Figure 35.

Additionally, the identical PySpark SQL results appear in Figure 36.

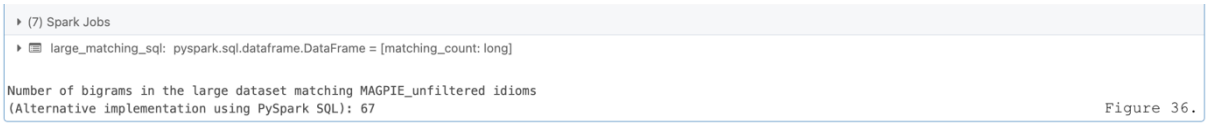


Figure 36.

Discussion, Exploration and Extension:

The identification of 67 unique bigrams from the `large` DataFrame that also appear in the `MAGPIE_unfiltered` idiom list indicates modest overlap between common two-word sequences in general text and known idiomatic expressions. This relatively small count is influenced by the precise nature of the analysis, reliant on exact string matches. This method only captures idioms within `MAGPIE` that are exactly two words long and appear identically in the Wikipedia bigrams, inherently missing longer idiomatic phrases. For a client, this indicates that while some common Wikipedia bigrams might be simple idioms, a more sophisticated approach would be necessary to identify a broader range of idiomatic expressions, particularly those longer than two words. Although the knowledge that these 67 bigrams are potential idioms is informative, it does not reveal which of these are most prevalent. To explore this, a further analysis was conducted to identify the 10 most frequent bigrams from these 67 matches, using their original frequencies as calculated from the Wikipedia dataset. This aimed to highlight prominent idiomatic phrases. The 67 matching unique bigrams were joined with the `large_bigram_counts` DataFrame (from Question 4, containing all Wikipedia bigram frequencies) on the `bigram` string. This result was then ordered by Wikipedia frequency descending, limited to the top 10, and the `bigram` and `count` columns were selected and aliased as `Bigram` and `Count` for display (Figure 37).

Figure 37.

The top 10 most frequent Wikipedia bigrams that are also present in the MAGPIE idiom list (Figure 38) include common, recognisable idiomatic expressions such as ‘on board’ (82 occurrences) and ‘game on’ (74 occurrences).

Figure 38.

Notably, the frequencies of these bigrams are substantially lower than those of the overall top raw bigrams identified in Question 4 (e.g. ‘of the’, 76,294 occurrences). However, this is expected, as specific idiomatic phases are naturally less frequent than common grammatical constructions. The lower frequencies indicate that these specific two-word idioms are not exceptionally high-frequency items in the broader context of all bigrams. The nature of these bigrams also differs significantly from the raw top 10, as this list consists of phrasal units rather than purely structural pairings. For a client, this curated list of frequently appearing, confirmed two-word idioms could be valuable for tasks such as identifying common figurative language, enriching lexicons for sentiment analysis, or for content generation requiring natural-sounding idiomatic phrases. However, as the method employed only captures two-word idioms that are an exact match for entries in MAGPIE, longer idioms or variations would be missed.

Question 6.

Assumptions:

The initial set of unique Wikipedia bigrams and their frequencies are accurately derived using the methodology established in Question 4, which includes word tokenisation via `F.split(F.col("sentence"), "\\s+")` and bigram formation using the `NGram(n=2)` transformer, followed by `groupBy("bigram").count()`.

A Wikipedia bigram is determined to be 'not in `MAGPIE`' if its exact string representation (case-sensitive and punctuation-inclusive) does not match any distinct idiom string from the `MAGPIE_unfiltered` DataFrame.

This specific comparison method is considered the basic pre-processing and analytical approach for this question, as the overall methodology yielded the expected results when tested on the `small.csv.gz` and `medium.csv.gz` datasets.

The specific sorting order utilised (`count`, then `bigram` string) and the use of `F.row_number()` directly determines which bigrams fall into the 2501-2510 rank window.

Implementation:

The solution for Question 6 leveraged methodologies from previous questions for initial data preparation. The `large_bigram_counts` DataFrame (Wikipedia bigrams and frequencies) was generated as per Question 4, and the distinct `MAGPIE` idiom list as per Question 5. The core logic then involved:

1. A `left_anti` join between `large_bigram_counts` and `MAGPIE_unfiltered_idioms` (on `F.col("bigram")==F.col("idiom")`) to isolate Wikipedia-only bigrams;
2. Ordering these bigrams by frequency descending, then `bigram` string ascending;
3. Assigning a unique rank using `F.row_number()` over a window defined by this order;
4. Filtering for ranks 2501-2510 inclusive;
5. A final `select()` to format the output columns as `Rank`, `Bigram` and `Count` (Figure 39).

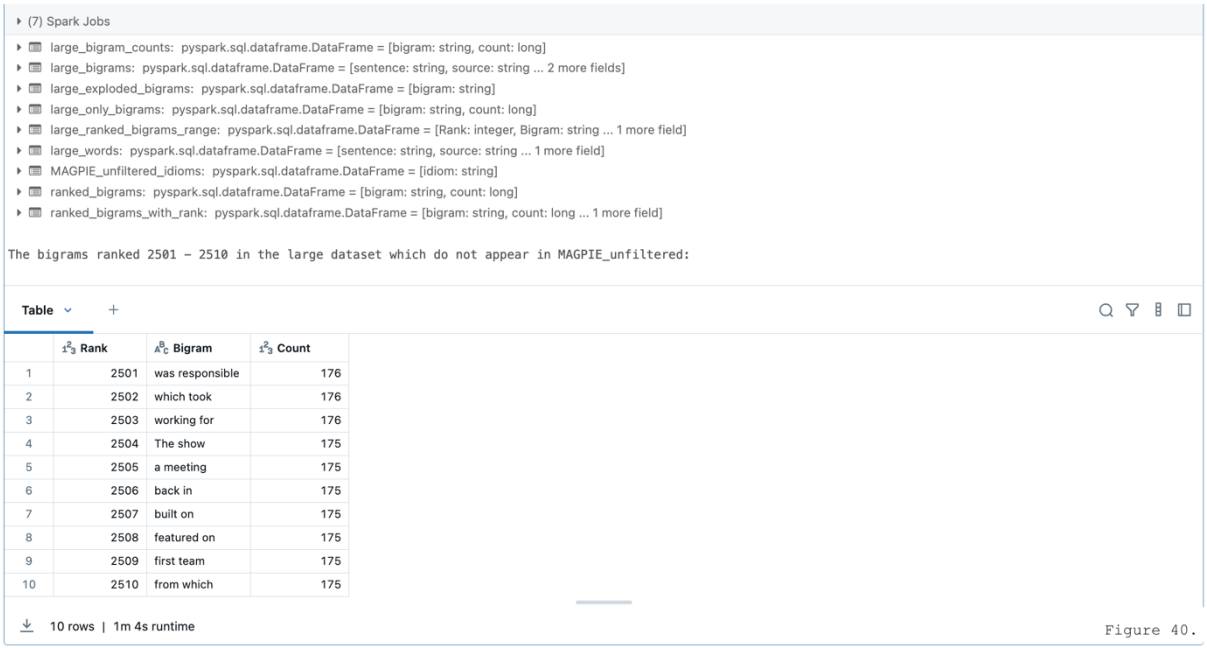
38 Python

```
large_words = large.withColumn("words", F.split(F.col("sentence"), "\\s+"))

large_bigram_transformer = NGram(n = 2, inputCol = "words", outputCol = "bigrams")
large_bigrams = large_bigram_transformer.transform(large_words)

large_exploded_bigrams = large_bigrams.select(F.explode(("bigrams")).alias("bigram"))

large_bigram_counts = large_exploded_bigrams.groupBy("bigram").count()

MAGPIE_unfiltered_idioms = MAGPIE_unfiltered.select("idiom").distinct()

large_only_bigrams = large_bigram_counts.join(MAGPIE_unfiltered_idioms, F.col("bigram") == F.col("idiom"), "left_anti",)

ranked_bigrams = large_only_bigrams.orderBy(F.col("count").desc(), "bigram")

ranked_bigrams_with_rank = (ranked_bigrams.withColumn("rank", F.row_number().over(Window.orderBy(
                                                                F.col("count").desc(),
                                                                F.col("bigram")
                                                                )
                                                        )
                                                )
)

large_ranked_bigrams_range = (ranked_bigrams_with_rank.filter((F.col("rank") >= 2501) & (F.col("rank") <= 2510)).select(
                                                                F.col("rank").alias("Rank"),
                                                                F.col("bigram").alias("Bigram"),
                                                                F.col("count").alias("Count")
                                                                )
)

print("\nThe bigrams ranked 2501 - 2510 in the large dataset which do not appear in MAGPIE_unfiltered:\n")
large_ranked_bigrams_range.display()
```

Figure 39.

Result:

The PySpark code identified the 10 bigrams ranked 2,501-2,510 present in the Wikipedia dataset but not in the `MAGPIE` idiom list. This list includes phrases such as 'was responsible' (rank 2,501, 176 occurrences), 'which took' (rank 2,502, 176 occurrences) and 'The show' (rank 2,504, 175 occurrences) (Figure 40).



(7) Spark Jobs

- large_bigram_counts: pyspark.sql.dataframe.DataFrame = [bigram: string, count: long]
- large_bigrams: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 2 more fields]
- large_exploded_bigrams: pyspark.sql.dataframe.DataFrame = [bigram: string]
- large_only_bigrams: pyspark.sql.dataframe.DataFrame = [bigram: string, count: long]
- large_ranked_bigrams_range: pyspark.sql.dataframe.DataFrame = [Rank: integer, Bigram: string ... 1 more field]
- large_words: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string ... 1 more field]
- MAGPIE_unfiltered_idioms: pyspark.sql.dataframe.DataFrame = [idiom: string]
- ranked_bigrams: pyspark.sql.dataframe.DataFrame = [bigram: string, count: long]
- ranked_bigrams_with_rank: pyspark.sql.dataframe.DataFrame = [bigram: string, count: long ... 1 more field]

```
The bigrams ranked 2501 - 2510 in the large dataset which do not appear in MAGPIE_unfiltered:
```

Table +

	Rank	Bigram	Count
1	2501	was responsible	176
2	2502	which took	176
3	2503	working for	176
4	2504	The show	175
5	2505	a meeting	175
6	2506	back in	175
7	2507	built on	175
8	2508	featured on	175
9	2509	first team	175
10	2510	from which	175

10 rows | 1m 4s runtime

Figure 40.

Discussion, Exploration and Extension:

Examining bigrams ranked 2501-2510 in the Wikipedia dataset reveals word pairings that are relatively uncommon but still occur with modest frequency, with these particular bigrams also confirmed not to be present in the `MAGPIE` idiom list. A key observation is their low and very similar frequency counts (176 and 175 occurrences), indicating a flattening of the bigram frequency distribution around rank ~2500, where many different bigrams share similar lower counts. These particular bigrams appear to be functional linguistic units rather than highly topical or unique descriptive phrases, consistent with their absence from the `MAGPIE` idiom list as they lack figurative meaning. For a client, this analysis demonstrates the long tail of the bigram distribution. Beyond the highly frequent items lies a vast vocabulary of unique or less common bigrams, many of which are standard structural pairings. Understanding this part of the frequency spectrum is valuable for tasks requiring deeper insight into general language patterns beyond just the most prominent or idiomatic expressions. The nature of these bigrams motivates further investigation into their contextual usage.

To better understand their typical context and actual usage, a further exploration was conducted involving retrieval and display of three example sentences from the original `large.csv.gz` dataset for each of the 10 bigrams. This involved collecting the target bigrams into a Python list, then iterating through it. For each bigram, the `large` DataFrame was filtered using `F.col("sentence").contains(target_bigram)` (case-sensitively), with three matching sentences selected and displayed (Figure 41).

```
40                                                                    Python
large_target_bigrams_list = [row.bigram for row in large_ranked_bigrams_range.select(F.col("bigram")).collect()]

print(f"\nExample sentences containing the bigrams ranked 2501 - 2510 in\nthe large dataset which do not appear in MAGPIE_unfiltered:\n")

for large_target_bigram in large_target_bigrams_list:
    print(f"Example sentences containing the bigram: '{large_target_bigram}'")

    large_example_sentences = large.filter(F.col("sentence").contains(large_target_bigram))

    large_example_sentences.select(F.col("sentence").alias("Sentence")).limit(3).display()
```

Figure 41.

Example sentences were retrieved for the 10 target bigrams (Figure 42), which included:

`Example sentences containing the bigram: 'was responsible'`

- "*However, her brother John, who **was responsible** for organising her dowry, was slow to do so.*"
- "*He **was responsible** for introducing a friend's poetry to Mr. Justice Talfourd (died 1854).*"
- "*He **was responsible** for expending and accounting for several million dollars as he acquired medical supplies and equipment for the Union Army, and distributed them to units throughout the country.*"

(17) Spark Jobs

large_example_sentences: pyspark.sql.dataframe.DataFrame = [sentence: string, source: string]

```
Example sentences containing the bigrams ranked 2501 - 2510 in
the large dataset which do not appear in MAGPIE_unfiltered:

Example sentences containing the bigram: 'was responsible'
```

Table

	Sentence
1	However, her brother John, who was responsible for organising her dowry, was slow to do so.
2	He was responsible for introducing a friend's poetry to Mr. Justice Talfourd (died 1854).
3	He was responsible for expending and accounting for several million dollars as he acquired medical supplies and equipment for the Union Army, and distributed them to units throughout the country.

3 rows | 44.70s runtime

`Example sentences containing the bigram: 'which took'`

Table

	Sentence
1	The four additional domes were added during renovations which took place between 2008 and 2016.
2	He returned back to Divizia A football during his second spell at Petrolul in the 1999–2000 season in which he earned two historical victories against Steaua București, a 5–1 at home and a 4–1 on the Ghencea stadium, also a 4–2 home victory against Mircea Lucescu's Rapid București who were the title holders, afterwards going to coach in the lower leagues for a second spell at Midia, later at Cimentul Fieni, Chindia Târgoviște and for a third spell at Plopeni, retiring after a third spell at Petrolul which took place from July until December 2004.
3	The 1987 Tolly Ales English Professional Championship was a professional non-ranking snooker tournament, which took place in February 1987 in Ipswich, England.

3 rows | 44.70s runtime

`Example sentences containing the bigram: 'working for'`

Table

	Sentence
1	At least once Isabella wrongly stated the light direction to the artists working for her and she often sent changed her mind about the subjects and compositions.
2	Gauri Maulekhi started working for People For Animals in Lucknow in 1995 as a volunteer, where she played a vital role in setting up the first animal shelter in the city.
3	He is now working for Kumar Mangat.

3 rows | 1m 4s runtime

`Example sentences containing the bigram: 'The show'`

Table

	Sentence
1	The show was based on Braly's childhood trips to Bangkok, Thailand.
2	The show was officially aired and broadcast online on March 17, 2018 on iQiyi.
3	The show was transformed when the writers decided to limit the storytelling, with the exception of the opening scene of the first episode, to the perspective of the eight characters.

3 rows | 1m 4s runtime

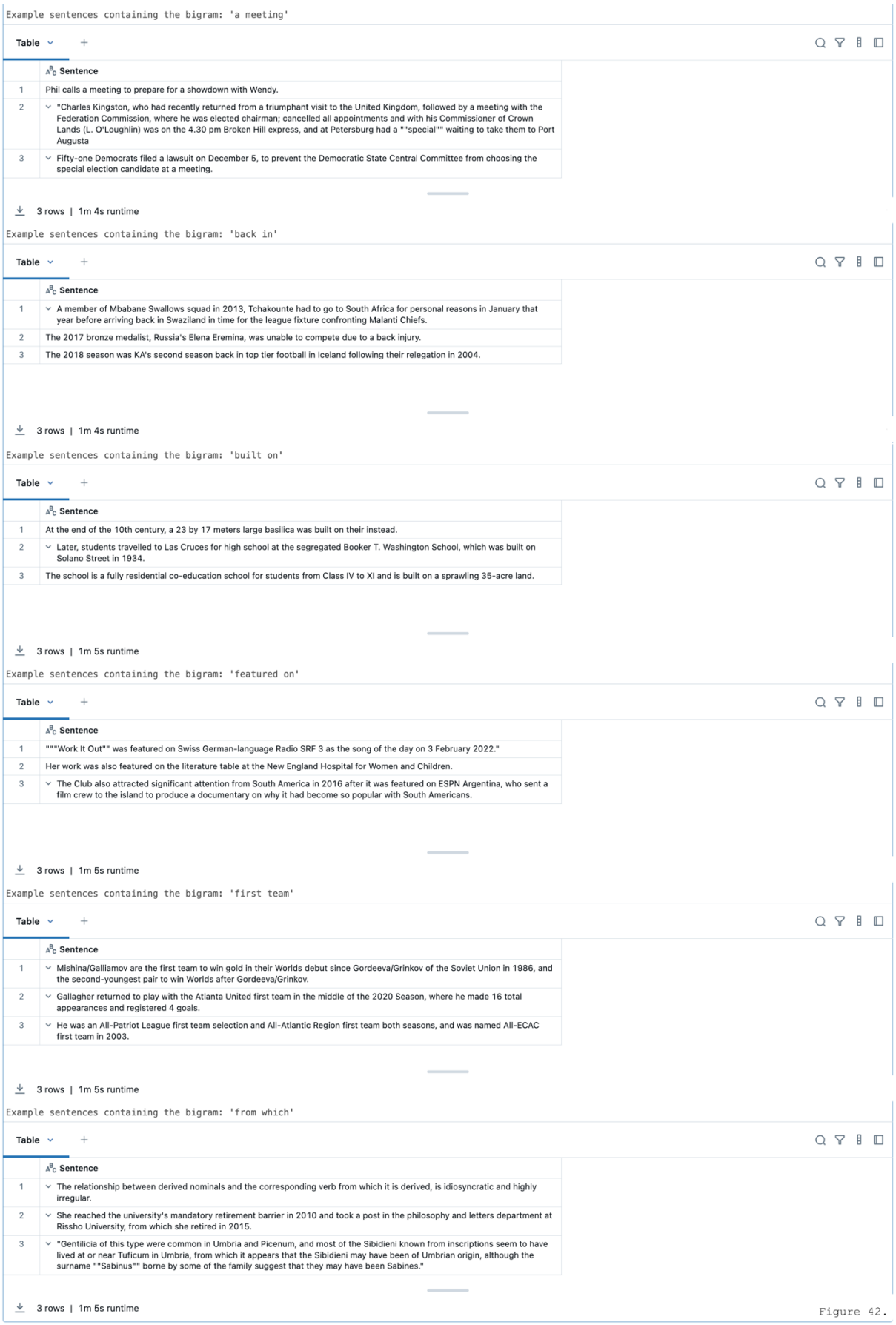


Figure 42.

Reviewing these bigrams within the context of examples of their original sentences provides valuable clarification on their typical usage. This contextual information helps to confirm their non-idiomatic nature and to better understand why these particular bigrams have their observed frequencies. For a client, this level of detail moves beyond simple frequency counts, offering a glimpse into actual linguistic usage. This

can be useful for fine-tuning phrase extraction for specific domains, understanding nuanced language, or validating the literal usage of recurring word pairs. This might be important for tasks such as knowledge base construction or information retrieval.